Darpa/Navy Contract No. N00014-92-J-1809

ControlShell: A Real-Time Software Framework

Quarterly Progress Report — July - September, 1993

P.I.'s: Prof. J.C. Latombe, Prof. R.H. Cannon, Jr, Dr. S.A. Schneider

Executive Summary

We are creating a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective is being met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research concentrates on three key areas:

- Building an innovative, powerful real-time software framework,
- Implementing new distributed control architectures for intelligent mechanical systems, and
- Developing distribution architectures and new algorithms for the computationally "hard" motion planning and direction problem.

Perhaps more importantly, we are working on the *vertical integration* of these technologies into a powerful, working system. It is only through this coordinated, cooperative approach that a truly revolutionary, usable architecture can result.

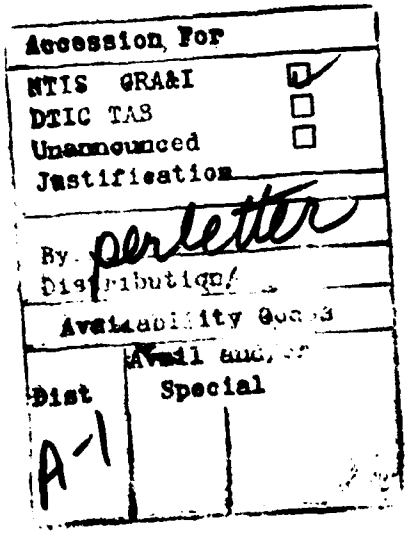
Summary of Progress

This section highlights some of our achievements for this quarter. During this period, we have:

- Added query capability and reliable updates to the Network Data Delivery Service.
- Released the first version of the graphical Finite-State Machine (FSM) Editor to our application users.
- Completed an initial implementation of the graphical Data-Flow Editor (DFE) tool for building complex control systems using a block-diagram paradigm.
- Completed an initial class design for the C++ implementation of the real-time ControlShell code.

- Built a 3-D simulator for our two-arm robot system.
- Developed an adaptive sensor fusion algorithm for matching our visual and kinematic sensor sets.
- Designed our major proof-of-concept demonstration.
- Investigated fast planning algorithms by distribution over problem approximation.
- Developed an efficient on-line algorithm for constructing the C-space for a dual-arm SCARA-type robotic system.
- Extended the path planner to consider the case of multiple tasks with multiple objects simultaneously.
- Implemented landmark-based navigation with an experimental mobile robot.

Our research is progressing according to schedule.



Chapter 1

Introduction

The goal of this research project is to build a new paradigm for building and maintaining complex real-time software systems for the control of moving mechanical systems. This objective is being met through the *simultaneous* development of both a powerful software environment and cogent motion planning and control capabilities. Our research concentrates on three areas:

- Building an innovative, powerful real-time software development environment,
- Implementing a new distributed control architecture, and using it to deftly control and coordinate real mechanical systems, and
- Developing a computation distribution architecture, and using it to build on-line motion planning and direction capabilities.

We believe that no technology can be successful unless proven experimentally. We are thus validating our research by direct application in several disparate, real-world settings.

This concurrent development of system framework, sophisticated motion planning and control software, and real applications insures a high-quality architectural design. It will also embed, in *reusable* components, fundamental new contributions to the science of intelligent motion planning and control systems. Researchers from our three organizations, the Stanford Aerospace Robotics Laboratory (ARL), the Stanford Computer Science Robotics Laboratory (CSRL), and Real-Time Innovations, Inc. (RTI) have teamed to cooperate intimately and directly to achieve this goal. The potential for advanced technology transfer represented by this cooperative, vertically-integrated approach is unprecedented.

Framework Development This research builds on an object-oriented tool set for real-time software system programming known as *ControlShell*. It provides a series of execution and data interchange mechanisms that form a framework for building real-time applications. These mechanisms

are specifically designed to allow a component-based approach to real-time software generation and management. By defining a set of interface specifications for inter-module interaction, ControlShell provides a common platform that is the basis for real-time code exchange and reuse.

Our research is adding fundamental new capabilities, including network-extensible data flow control and a graphical CASE environment.

Distributed Control Architecture This research combines the high-level motion planning component developed by the previous effort with a deft control system for a complex multi-armed robot. The emphasis of this effort is on building interfaces between modules that permit a complex real-time system to run as an interconnected set of distributed modules. To drive this work, we are building a dual-arm cooperative robot system that will be able to respond to high-level user input, create sophisticated motion and task-level plans, and execute them in real time. The system will be able to effect simple assemblies while reacting to changing environmental conditions. It combines a world modelling system, real-time vision, task and path planners, an intuitive graphical user interface, an on-line simulator, and sophisticated control algorithms.

Computation Distribution Architecture This research thrust addresses the issues arising when computationally complex algorithms are embedded in a real-time framework. To illustrate these issues we are considering two particular problem domains: *object manipulation by autonomous multi-arm robots* and *navigation of multiple autonomous mobile robots in an incompletely known environment.* These two problems raise a number of generic issues directly related to the general theme of our research: motion planning is provably a computationally hard problem and its outcomes, motion plans, are executed in a dynamic world where various sorts of contingencies may exist.

The ultimate goals of our investigation are to both provide real-time controllers with on-line motion reactive planning capabilities and to build experimental robotic systems demonstrating such capabilities. Moreover, in accomplishing this goal, we expect to identify general guidelines for embedding a capability requiring provably complex computations into a real-time framework.

Chapter 2

ControlShell Framework Development

This section describes our progress in developing the ControlShell framework and underlying architecture. Two fundamental extensions to ControlShell are being pursued:

- Distributed information sharing paradigms, by Gerardo Pardo-Castellote and Stan Schneider.
- Graphical Computer Aided Software Engineering (CASE) environments, by Stan Schneider and Vince Chen.

2.1 Distributed Information Sharing Paradigms: NDDS

A considerable effort has been made to prepare NDDS for our target release date in the second quarter of 1994. The code has been cleaned and documented thoroughly, detailed manual pages have been written for all the interface functions and two demonstration programs have been developed to be distributed with NDDS as implementation examples.

During the previous quarterly report we have motivated the need for providing support for reliable updates. In essence there is certain kind of data for which reliable, in-order distribution is a must.

During this quarter we designed and incorporated into NDDS a mechanism to support reliable updates. It is not obvious at all how to provide clear semantics to the idea of "reliable message delivery" in a situation where there may be multiple anonymous producers and consumers of the same data instances. More so because the NDDS model has purposely isolated producers from consumers and neither one has any means of knowing how many (if any) other producers or consumers there are at any one time for any particular data instance.

In particular a good model should address the following issues:

- Who specifies reliability: the producer, the consumer or both?
- What is specified as reliable? All the productions of a producer? The consumptions of a consumer? A particular data instance? A particular update?
- What is the meaning of reliability? What is considered to be a failure? For example, lets say the user specifies a particular update to be delivered reliably and issues such an update, however, there are no consumers that have subscribed to that particular data. Is this a failure? Beyond this, what if there are some subscribers and some of them receive the update and others don't. Is this second scenario a failure?
- How and when does the application get notified of failures?
- Are reliable productions of the same data delivered in order? Should they be?

In accordance with the philosophy of NDDS we have attempted to provide a consistent, well-defined model that incorporates the essential functionality for a distributed-control type application. The end user may well choose to write a custom interface layer over NDDS to customize its behavior to its own specific needs.

We have provided an API in which, under normal operation (i.e. when no failures occur), reliable productions look just like any other production to the user. Only when failures do occur, is the user notified so that appropriate action can be taken.

We have chosen the producer to be the one to specify which data instances should be delivered reliably. Reliable productions can be specified on an instance-by-instance basis. We have also provided a mechanism for the user to install custom methods that will be invoked if an error arises. In this manner, the user can tailor the error handling to each individual situation.

The fact that some of NDDS's applications might be mission-critical suggests that NDDS should enforce, or at least provide, mechanisms for higher levels of reliability than just communication failures. Some of the errors listed below would not be considered as errors in some specific applications and the user may wish to mask them out. NDDS considers the following error conditions:

- No subscribers to a reliable update. This error arises when the user specifies an update to be delivered reliably and there are no subscribers to the update. Under some circumstances, this may be a serious failure. The update could be sending a shutdown signal to some remote equipment; the fact that there are no subscribers indicates that the equipment is not listening for commands. This could be due to either a failure in the remote software, in the communications link, or in the equipment itself. In any case the sender should be notified of this fact immediately.
- Deadline expired. A reliable update was sent and none of the subscribers acknowledged it within a specified deadline.

- Pending update. A second reliable update is attempted while a previous update to the same host is still pending. The pending update may be for the same or some other data instance from any producer in the sending host. This would indicate either a communication failure, or attempts to send at a faster rate than can be accommodated by the acknowledgement latency of the system. Acknowledgement latency could be reduced by using a windowing scheme, allowing several updates to be simultaneously pending.

- Stronger producer. A reliable production is attempted while the persistence of a previous update of a (reliable or unreliable) producer hasn't expired. This is an indication that a local stronger producer is in operation and the weaker reliable production will not be delivered.

If any of the above error conditions occurs the user-provided error routine gets called with the appropriate error code. Notice that it is still possible that the consumer will never receive the data and the reliable producer won't be notified. This situation arises when the remote receiver gets the update and acknowledges it, but the consumer doesn't get an update for a specific instance because there has been a previous update by a remote stronger producer which hasn't expired. There are other transient conditions that result in similar behavior. We feel this isn't a mayor problem. After all, if the producer has to be sure that its update accomplishes the desired action, then there is no other choice than to get confirmation from the consumer himself. This must be the responsibility of the application software. NDDS can only ensure that the message is delivered reliably and that there is somebody waiting for it. Whether the receiver decides to ignore it or not is beyond the mission of NDDS.

2.2 ControlShell CASE Environment

Excellent progress was made in this quarter in the area of developing ControlShell's graphical CASE environment.

The graphical Finite-State Machine (FSM) Editor was released for testing this quarter. The functionality of this tool was described in the last report.

Our work this quarter focused on the development of the graphical Data-Flow Editor (DFE). The DFE tool allows a user to build ControlShell systems by connecting functional blocks.

Component-Based Design ControlShell is designed to encourage *component-based* design. As such, ControlShell provides interface definitions and mechanisms for building real-time code modules called *components*. ControlShell systems are built from combinations of these components.

The component is the fundamental unit of reusable data-flow code in ControlShell. Components consist of one or more *CSSampleModules* derived from CSModules. Sample modules have several pre-defined entry routines, including:

Routine	When executed
execute	Once each sample period
stateUpdate	After all executes are done
enable	When this module is made active
disable	When it is removed from the active list
startup	When sampling begins
shutdown	When sampling ends
timingChanged	When the sample rate changes
reset	When the user types "reset", or calls CSSampleReset
terminate	When the module is unloaded

Thus, a motor driver component might define a startup routine to initialize the hardware, an execute routine to control the motor, and a shutdown routine to disable the motors if sampling is interrupted for any reason. In addition, if any of its parameters depend on the sampling rate, it may request notification via a timingChanged method. By allowing components to attach easily to these critical times in the system, ControlShell defines an interface sufficient for installing (and therefore sharing) generic sampled-data programs.

An extensive library of pre-defined components is provided with the system, ranging from simple filters and controllers to complex trajectory generators and motion planning modules.

The Component Editor A graphical tool called the *component editor* (CE) assists the user in generating new components and specifying their data-flow interactions. The Component Editor defines all the input and output data requirements for the component, and creates a data type for the system to use when interacting with the component. The tool contains a code generator; it automatically generates a description of the component that the Data-Flow Editor can display (see below), and the code required to install instances of the component into ControlShell's run-time environment. The component editor was a part of the original non-graphical ControlShell system. However, it is currently out of date; it currently does not support the DFE editor fully, nor the full component structure. It will undergo a major design revision in the next quarter.

The Data-Flow Editor This quarter, we have completed our initial implementation of the Data-Flow Editor (DFE). The DFE is used to connect components into a complex system.

The DFE reads description files produced by the component editor, and then allows the user to connect components in a friendly graphical environment. It allows specification of all the data connections in the system, as well as reference inputs—gains, configuration constants and other parameters to the individual components. An example session is depicted in Figure 2.1.

The DFE Editor allows the user to describe the controls system using block diagrams and signals. Each component, defined by the *ControlShell* Component Editor and user code, is represented by

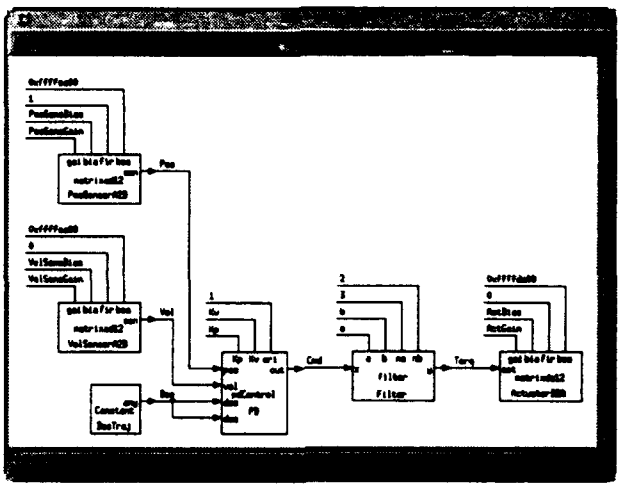


Figure 2.1: **Data-Flow Editor**

The data-flow editor builds collections of components into an executing system.

a block with clearly marked input, output, and reference "pins". The user connects the pins of the components using named signal lines, thereby defining the data flow in the system.

The DFE Editor generates files that are parsed by the run-time system to determine the execution order of each component. It will also allow the user to specify groups of components to be enabled and disabled at run-time, replacing the original ControlShell's configuration manager with a much friendlier and more powerful graphical interface. This will be a prime focus of our work in the next quarter.

The DFE Editor is written in C++ for the MOTIF/X environment. A detailed description is not provided here; the preliminary User's Manual is attached.

2.2.1 Object-Oriented Design

This quarter, we also began the redesign of ControlShell to take advantage of the object-oriented features of C++. ControlShell's already modular and object-oriented design makes this transition relatively easy. This section summarizes our design.

2.2.1.1 Module Classes

CSModules The ControlShell run-time architecture is based on a simple concept: the *CSModule*. A CSModule is a named routine with pointers to data already bound to it. The system can find and execute any CSModule at any time without needing to supply the CSModule with its data.

The `CSModuleClass` is the abstract base class of all ControlShell execution modules. It binds a name to an execution routine (pure virtual function). Instance classes are expected to define the actual execution code as well as any data needed for execution.

CSSampleModules A *CSSampleModule* extends upon the CSModule by binding multiple routines, each of which is executed at well-defined times. Additionally, each CSSampleModule contains lists of input and output signal dependencies allowing CSSampleModules to be sorted to determine the order of execution.

`CSSampleModuleClass` is also an abstract base class derived from `CSModuleClass`. It binds other execution routines that can be expected of a module that executes on a sample list. These additional methods include `stateUpdate`, `enable`, `disable`, `startup`, `shutdown`, `timingChanged`, `terminate`, and `reset`, etc. as detailed above. Instance classes only need to define and implement the needed methods. If not defined, they default to null routines.

Component Classes *CSComponents* are further derived from CSSampleModules, providing methods to print the data structure bound to the component—in formats for human or machine consumption. The `CSComponentClass`, derived from `CSSampleModuleClass`, serves as the base class for all ControlShell components.

Each component will be implemented as a separate derived class. The derived class definition will be automatically generated from the Component Editor's description of the component's data flow requirements. Thus, the data structure and skeleton code for the required methods for the component is automatically built for the user. The generated code also includes methods to parse data-flow description files (generated by the DFE editor) and to instance new objects that implement components.

2.2.1.2 Execution List Classes

CSSampleLists ControlShell uses *CSSampleLists* to manage the execution of CSSampleModules. A CSSampleList contains a list of registered CSSampleModules that are to be sequentially executed. Based on the trigger that starts the execution sequence, the CSSampleList determines which of the CSSampleModule's routines to run. For example, at "Sample" time, every (enabled) module's `execute` routine is executed, then every module's `stateUpdate` routine is executed.

`CSSampleListClass` is the base class that provides facilities for registering CSSampleModules and methods that can be called at these "well-defined" times to execute the proper module routines. The `CSSampleListClass` and its subclasses are internal to ControlShell and are not meant to be directly manipulated by the user.

CSSampleHabitats Finally, a *CSSampleHabitat* is derived from CSSampleList to provide a named sampled-data environment. A CSSampleHabitat encapsulates all the information and defines all the interfaces required for sampled-data programs to co-exist. It also contains routines to control the sampling process and timing source. For example, a module installed into a sample habitat can query its clock source and sample rate, start and stop the sampling process, etc.

Each sample habitat contains an independent task that executes the sample code. The task is clocked by the periodic source (such as a timer interrupt). Additionally, the execution order in a `CSSampleHabitatClass` is automatically determined by sorting the CSSampleModules (and their derived Component classes) according to their input and output dependencies.

The ControlShell structure described here is quite amenable for implementation using C++. The ControlShell class structure consists of a fairly shallow tree to allow users to develop ControlShell components quickly and painlessly, without having to dig through the inheritance tree. Moreover, the automatic code generation of the ControlShell Component Editor further shortens development time.

2.2.2 Configuration Management

During this quarter, we also made good progress in designing the configuration management capabilities of ControlShell.

Complex real-time systems often have to operate under many different conditions. The changing sets of conditions may require drastic changes in execution patterns. For example, a robotic system coming into contact with a hard surface may have to switch in a force control algorithm, along with its attendant sensor set, estimators, trajectory control routines, etc.

ControlShell's configuration manager directly supports this type of radical behavior change; it allows entire groups of modules to be quickly exchanged. Thus, different system personalities can be easily interchanged during execution. This is a great boon during development, when an application programmer may wish, for example, to quickly compare controllers. It is also of great utility in producing a multi-mode system design. By activating these changes from the state-machine facility (see previous reports), the system is able to handle easily external events that cause major changes in system behavior.

Configuration Hierarchy The configuration manager essentially creates a four-level hierarchy of module groupings. Individual sample modules form the lowest level. These usually implement a

single well-defined function. Sets of modules, called *module groups*, combine the simple functions implemented by single modules into complete executable subsystems.

Each module group is assigned to a *category*. One group in each installed category is said to be *active*, meaning its modules will be executed. Finally, a *configuration* is simply a specification of which group is active in each category.

Example As a simple example, consider a system with two controllers: a proportional-plus-derivative controller named "PD", and an optimal controller known as "LQG". Suppose the PD controller requires filtered inputs, and thus consists of two sample modules: an instance of the *PDControl* component and a filter component. These two components would comprise the "PD" module group. The "LQG" controller module group may also be made up of several components. Both of these groups would be assigned to the category "controllers".

The user (or application code) can then easily switch controllers by changing the active module group in the "controller" category.

Now suppose further that the controllers require a more sophisticated sensor set. A category named "sensors" may also be defined, perhaps with module groups named "endpoint" and "joint". The highest level of the hierarchy allows the user to select an active group from each category, and name these selections as a *configuration*. Thus, the "JointPD" configuration might consist of the "joint" sensors and the "PD" controller. The "endptLQG" configuration could be the "endpoint" sensors and the "LQG" controller.

Category and Group Specification This subdivision may seem complex in these simple cases. However, it is quite powerful in more realistic systems. It has been shown to be quite natural in our experimental applications. Our work this quarter has focused on implementing this structure within the new graphical editing tools.

Assigning modules to groups and groups to categories is made quite simple with the ControlShell graphical DFE editor's "configuration definition" window, shown in Figure 2.2. New categories are added with the click of a button. To create a module group, the user simply names a group, and then clicks on the modules in the data-flow diagram that should belong to that group. The blocks are color-coded to relate the selections back to the user.

During the next quarter, we will design and implement the classes required on the real-time system to manage these configurations.

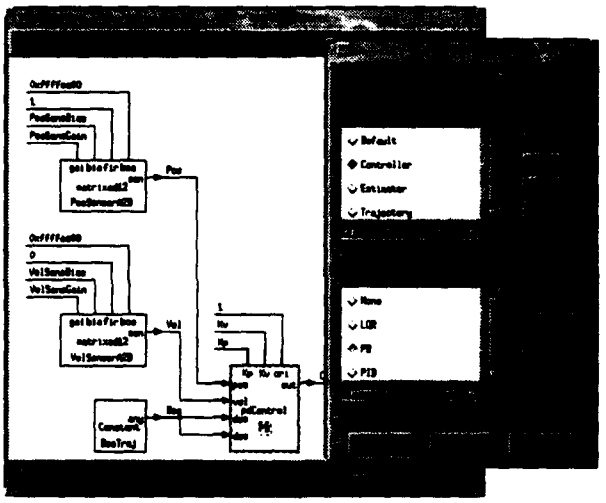


Figure 2.2: Configuration Definition

Configurations are easily defined within the DFE graphical interface.

Chapter 3

Distributed Control Architectures and Interfaces

This section covers our research in software architectures, communication protocols and interfaces that will advance the state-of-the-art in the prototyping-development-testing cycle of high-performance distributed control systems. These interfaces will be implemented within the framework described in Chapter 2. The results of this research will be applied to the vertical integration of planning and control and demonstrated by executing a set of challenging tasks on our two-armed robot system.

There are three main thrusts to this research:

- Development of inter-module interfaces for distributed control systems, by Gerardo Pardo-Castellote.
- Development of a control methodology capable of executing high-level commands, by Gerardo Pardo-Castellote, Tsai-Yen Li, and Yotto Koga.
- Hardware development and experimental verification, by Gerardo Pardo-Castellote and Gad Shelef.

This quarter, we focused on the control methodology as well as the development of a three-dimensional robot simulator suitable for testing with the planner modules.

3.1 Inter-Module Interfaces

This quarter we have developed a full 3-D graphical simulator for the pair of cooperating robots. The goal of the simulator is to allow rapid testing and debugging of all the system modules without

the use of the actual robots (the use of the robots is much more time consuming and risks damaging them). Since all the interfaces are built on NDDS, none of the remaining modules is aware that they are sending commands and receiving updates from the simulator instead of the actual robot. In the past, a 2-D simulator was used. This simulator was effective in testing the collision-free characteristics of the paths in the plane, but it neglected the third dimension (which would have to be tested directly in the real system). The 2-D simulator didn't have any knowledge of the robot dynamics (i.e. it was limited to following the via points with constant time-steps between them). The new 3-D version uses 3-D graphics built using the PHIGS libraries to represent full motions in 3-D space and incorporates the Via-Point Trajectory generation algorithm with the actual robot dynamics to simulate trajectory following with the same time profiles as the physical robot.

The task interface described in the previous report allows the user-interface module to send task commands to the planner. One of these tasks consists of placing multiple objects at specified locations. We have modified the 3-D graphical user interface to allow the user to specify these tasks interactively. The user selects multiple objects and drags their ghost images around the screen to indicate the desired goal locations for these objects. Once the task is selected, it can be sent to the planner with a simple mouse click.

3.2 Control Methodology Development

Two problems were identified last quarter: initial power-up calibration without hard stops and fusing the kinematic and vision information to produce an estimate of the tool's position that is adequate both for high bandwidth control and to acquire vision-sensed objects.

We have addressed both issues simultaneously with an adaptive scheme: once the kinematic parameters of the arms have been identified as described in the previous quarterly report, the only parameters that need to be identified in power-up are the initial angular offsets of the first two joints (shoulder and elbow). In addition, the small differences between kinematic values and vision values for the tool coordinates (which where of the order of 3 mm) can be mapped, using the inverse Jacobian to "equivalent" errors in the angular offsets. These facts are combined to develop the estimator of the "angular offsets" that would produce perfect correspondence between kinematic and vision coordinates shown in figure 3.1.

Figure 3.2 shows the performance of the adaptation system. The two fundamental problems are solved: The resulting signal isn't affected by the delay of the vision system (about 50 ms) and the signal converges to the vision value in steady state (in about 1 sec.)

Figure 3.3 illustrates the identification of initial offsets after the encoders have been powered-up with the arms places at a location far from the typical power-up configuration.

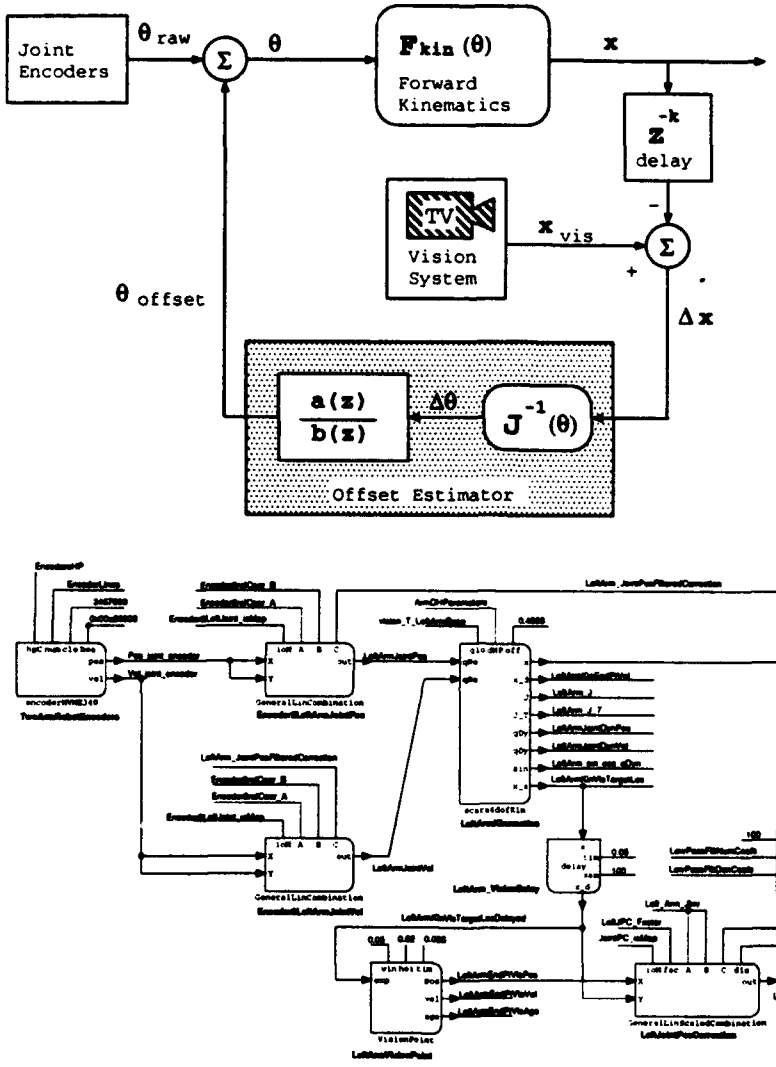


Figure 3.1: On-Line estimator of initial angular offsets

The (shoulder and elbow) joint angles produced by incremental joint encoders need to be combined with the initial angular offsets that correspond to the arm power-up location. In addition, these offsets are used to compensate for the (configuration-dependent) small differences between kinematics and vision coordinates. The offset estimator maps the Cartesian error between kinematic and vision coordinates through the inverse Jacobian to obtain a corresponding joint-space error. This error is run through a filter (essentially an integrator followed by a low pass filter) to produce an estimate of the angular offsets. This process is illustrated in the leftmost figure. The figure on the right shows the ControlShell implementation using generic software components.

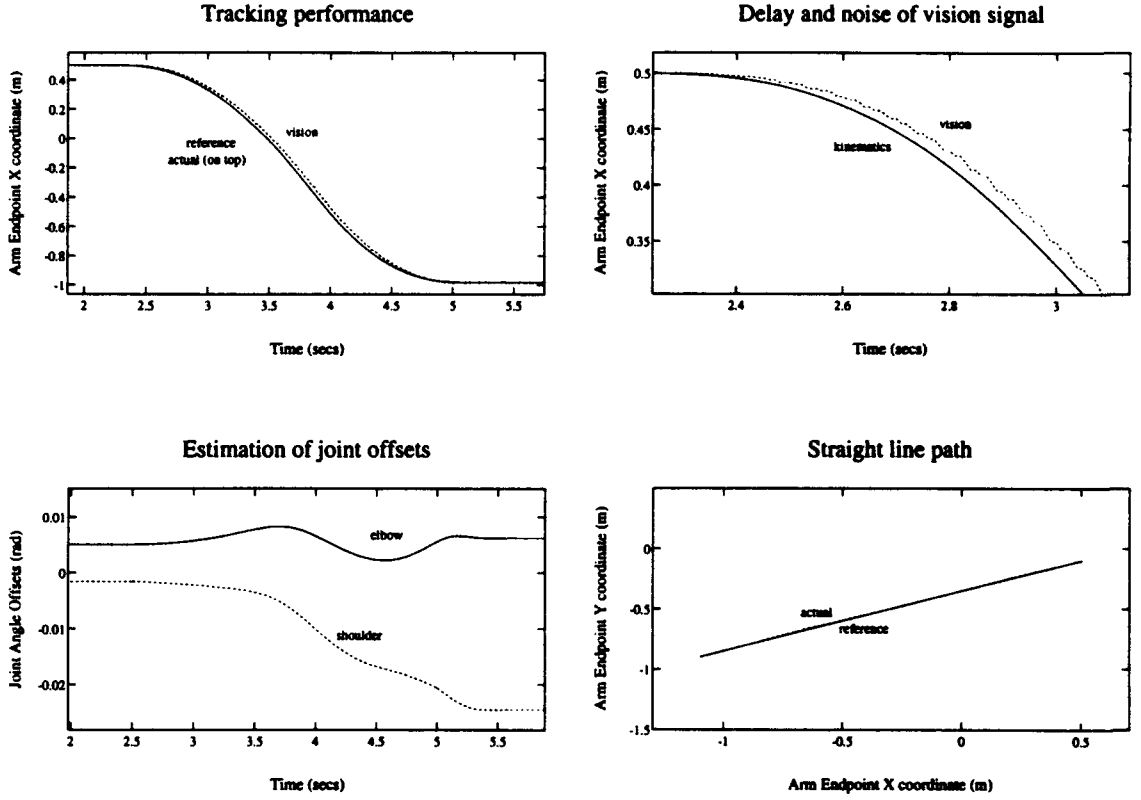


Figure 3.2: **Performance of On-Line angular-offset estimator**

The top-left figure shows both kinematic and vision coordinates during a straight-line slew (bottom-left figure). The vision signal isn't suitable for high-performance control: it is quite noisy and has a delay of about 50 ms due to the processing overhead (detail in top-right figure). The on-line estimator adapts the values of the shoulder and elbow angular offsets (bottom-left figure) so that at the end of the trajectory the kinematic and vision coordinates match again.

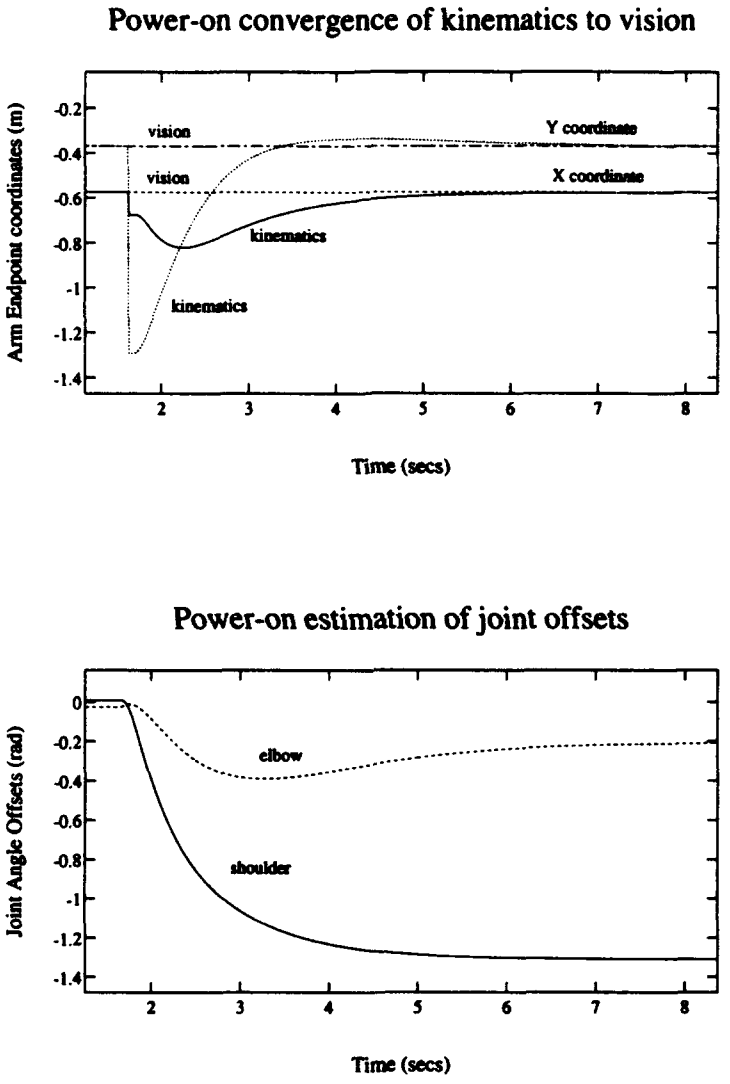


Figure 3.3: **Adaptation to power-up configuration**

When the incremental encoders are powered-up, it becomes necessary to identify the angular offsets that correspond to the location of the arms. In this figure the arms are held at an unusual location when the encoders are powered-up. As a result the initial kinematic location is very different from the (true) vision location. As the angular offsets are identified (right figure) the kinematic coordinates converge to the vision coordinates (left figure).

3.3 Hardware Development and Experiments

We plan to perform a demonstration which will both serve as proof-of-concept and focus for our research.

The user will select an assembly from the GUI, the required objects for this assembly will come down a conveyor at any time, and in any order, orientation, speed, etc. These objects may come around several times as required (the conveyor can be viewed as a part-supplier). The planner will direct the arms to acquire the objects from the conveyor. Some of the objects will be designed to require both arms to be manipulated, others will be graspable with a single arm. Next, the system will deliver the objects to their final assembly positions, clearing any workspace obstacles during the motion.

This project touches several important issues. Not only would it demonstrate the feasibility of assembly without fixturing, scheduling or sequencing, but also it will show the ability to interweave on-line planning of the sequencing operation, collision-free path computation and re-grasping operations with the real-time issues of trajectory tracking, capturing an object off a moving conveyor, etc.

To achieve this, at least three operating layers or regimes must be carefully combined: certain operations (e.g. planning) are more appropriately done with the aid of the powerful computers that are not likely to be part of the robot controller. Other operations, such as the final acquire from the moving conveyor, require high-bandwidth sequencing and event reactions and are therefore better done by the real-time system. Finally, others such as the underlying control algorithms run periodically at fixed high-rate loops and require the structure of a classical control-loop. The ability to integrate all these software modules to achieve the goal will be an excellent demonstration of the power of our approach.

Chapter 4

On-Line Computation Distribution Architectures

This research addresses technical issues arising when computationally complex algorithms are embedded in a real-time framework. To illustrate these issues we consider two particular problem domains: *object manipulation by autonomous multi-arm robots* and *navigation of multiple autonomous mobile robots in an incompletely known environment.* These two problems raise a number of generic issues directly related to the general theme of our research: motion planning is provably a computationally hard problem and its outcomes, motion plans, are executed in a dynamic world where various sorts of contingencies may happen.

The ultimate goal of our investigation, concerning the two problem domains mentioned above, is to both provide real-time controllers with on-line motion reactive planning capabilities and build experimental robotic systems demonstrating such capabilities. Moreover, in accomplishing this goal, we expect to elaborate general guidelines for embedding a capability requiring provably complex computations into a real-time framework.

This quarterly report covers work done towards this goal during the period of July, August, and September 1993. During this period, our work addressed on the following areas:

1. Distribution of Path Planning, by Tsai-Yen Li.
2. Parallelization of Path Planning, by Lydia Kavraki.
3. New Methods for Fast Path Planning, by Tsai-Yen Li.
4. Multi-Arm Manipulation Planning in 3D, by Yotto Koga.
5. Experiments in Manipulation Planning, by Tsai-Yen Li and Yotto Koga.
6. Landmark-Based Mobile Robot Navigation, by Anthony Lazanas, Byung-Ju Kang and Ken Tokusei.

7. Mobile Robot Navigation Toolkits, by Craig Becker, Mark Yim and David Zhu.

8. Multi-Mobile Robot Simulator, by Craig Becker and David Zhu.

Areas 1 through 5 are mainly related to the first problem domain, i.e., *object manipulation by autonomous multi-arm robots.*

Areas 6 through 8 are mainly related to the second problem domain, i.e., *navigation of multiple autonomous mobile robots in an incompletely known environment.*

Participating Ph.D. Students: Craig Becker, Lydia Kavraki, Yotto Koga, Anthony Lazanas, Tsai-Yen Li, Mark Yim.

Participating Master Students: Ken Tokusei, Byung-Ju Kang.

Participating Staff: David Zhu.

4.1 Distribution of Path Planning

We continue our research in applying the distribution methodology to the scenario of our final demonstration. In the last quarter report, we showed that we can distribute planning over time, and over several processors. In this quarter, we investigate the possibility of distributing the problem along the axes of problem approximation.

The distribution of planning over problem approximation can be used in searching for a delivery path (the path to deliver an object being grasped). Each object has three degrees of freedom in our settings and needs to avoid collisions with obstacles in the work space. A valid path is a collision-free path in the 3D configuration space (C-space) of the object. We compute the minimal enclosing circle and the maximal enclosed circle for the objects from the grasp point and then grow the obstacles according to the radiuses of these circles. The radiuses of these circles are called the *maximal ring* and the *minimal ring* respectively while the corresponding space after growing the obstacles are called the *impossible space* and the *safe space* [1]. Notice that the space with grown obstacles is still a 2D space which is much smaller than the size of the original space. If there exists a path in the safe space, then there must exists a path in the 3D object C-space. On the other hand, if there does not exist a path in the impossible space, then one can conclude that there is no path in the object 3D space. Therefore, we can distribute the planning for the delivery path over three different approximations, namely, searching in the original 3D C-space and the two reduced/projected spaces. If any useful information is returned from the search processes in the reduced spaces, one can either generate a valid path or conclude that there is no path without waiting for the search result of the original 3D C-space. Although in the worst case, we still need to search the whole 3D C-space, distribution over approximation increases our chance of finding a good solution much faster and therefore is more desirable for on-line application.

[1] Philippe Pignon, Optimal obstacle growing in motion planning for mobile robots, *IROS 91*, Osaka, Japan

4.2 Parallelization of Path Planning

During this quarter we started implementing the preprocessing phase of the path planning approach we have proposed in our previous report. Our intention is to gradually build a system which after a rather costly preprocessing of the configuration space (C-space) of the robot, will be able to solve path planning problems from any initial to any final configuration in this space very fast. We focus from the beginning on articulated robots with many degrees of freedom and our target is to solve problems that can not be solved, or take long to solve, with current techniques.

The preprocessing phase of our approach consists of two parts: (a) the generation of a large number of collision-free random configurations in the C-space of the robot and (b) their interconnection to a network, where the configurations are the nodes and an edge between two nodes denotes that a path connecting these nodes has been established. Depending on the connectivity of the robot's free space this network might contain one or several connected components. During path planning we hope to easily connect the beginning and the final configuration of the robot to two nodes A and B in the same connected component of our network and then search the network for a sequence of edges that connects A and B.

Part (a) of the preprocessing has been implemented. To achieve the generation of a large number (in the order of thousands) of collision-free random configurations in a short time, fast collision and self-collision procedures are required. This is because a very small percentage of the C-space of the robot is free (typically less than 1% in our examples). Also, care is taken to produce a rather uniform distribution (for example, for an articulated robot with 10 degrees of freedom, dof, we choose each dof uniformly from its allowed range).

We have also implemented an initial version of part (b) of the preprocessing stage of our algorithm. This part is computationally expensive, since a large number of connections need to be performed. We choose to use very simple and fast path planners for these connections. However, these planners are weak, and may fail to connect configurations that seem easily connectable. This is the reason why we are selective in which configurations we attempt to connect with our simple planner. We define a metric in the C-space and for each node x we sort all nodes according to increasing distance from x. The simple planner attempts to connect x to the M closest nodes, where M is a parameter. We store the information of whether two nodes can be connected with the simple planner used, but not the actual path, since the latter can be easily recovered. Examples of simple planners tried at this stage include the straight line in the multi-dimensional C-space of the robot and a planner that connects two nodes by advancing each joint along a straight line in the workspace connecting its initial and its final configuration.

After all configurations have been examined, the connected components of the resulting network are computed by a straightforward breadth-first search algorithm. In difficult cases (for example when the robot needs to go though a narrow passage and the configurations it can assume there are constrained), the simple planner of part (b) above may fail to produce a single connected component covering the free C-space of the robot. We believe that at this stage we can use a more involved path planner to connect the components. This planner will be expensive computationally but it

will be used only a few times.

During next quarter we plan to investigate the combination (and tradeoffs) of using very simple path planning techniques for producing a large number of easy connections and a more sophisticated planner for the connections that turn out difficult to achieve. We are interested in producing, in a reasonable time, a single connected component that contains most of the random configurations of part (a) and captures the connectivity of the free space as well as possible.

4.3 New Methods for Fast Path Planning

As mentioned in the last quarter report, efficiency is a crucial factor in bringing motion planners on-line. The geometric collision check between the robot and obstacles usually is the most time-consuming component of most motion planners. By building an explicit representation of the so called configuration space (C-space) one can greatly reduce the time for detecting collisions. In the previous reports, we presented an efficient method for constructing a 3D C-space using an FFT method for a free-flying object with obstacles in a 2D workspace. In this quarter, we focus on developing an efficient on-line algorithm for constructing the C-space for the dual-arm SCARA-type robotic system developed in the ARL. The result can be generalized for many other articulate manipulators.

For the dual-arm manipulator in the ARL, we assume that the obstacles for each arm are the links of the other arm. Although each arm has 4 DOF, we assume that the arms always move their grippers all the way up before moving so that we only need to consider the collisions between the two horizontal links of each arm. Building the C-space for the two arms essentially is to find the configurations that any of these links collide. This C-space is a 4D C-space (2D for each arm) consists of slices of 2D C-spaces corresponding to the configurations that one arm is fixed.

One way to compute the C-space obstacles is by discretizing the joint angle of each link and enumerating all possible discrete configurations to check for collision. By pre-computing the relative C-space between two links of the arms, one can speed up the detection of collision by some factors. However, with further observation, we can compute the arm C-space even more efficiently. We notice that the C-obstacle for one arm with the other arm fixed is a slanted cylindrical surface in the relative link-to-link C-space. Instead of enumerating every configuration and checking for collisions, one can extract a 2D C-space obstacles for two links directly from the surface in the link-to-link C-space. The final 2D arm C-space is formed by superimposing the C-space obstacles of the individual links. Since only memory copying and binary rotations are involved in this procedure, the computation of a 2D slice of C-space is extremely efficient.

Compared to the brute-force way of computing the C-space by enumerating every possible configuration and checking for collision, the new proposed method has a speedup of 70 and takes only 1 ms to compute for a 2D slice of arm C-space on a 28-mips workstation. The efficiency of this C-space construction enable us to compute each 2D slice of arm C-space on-line when it is needed. The overall performance of the planner is also greatly improved due to the efficiency of collision

checks.

4.4 Manipulation Planning

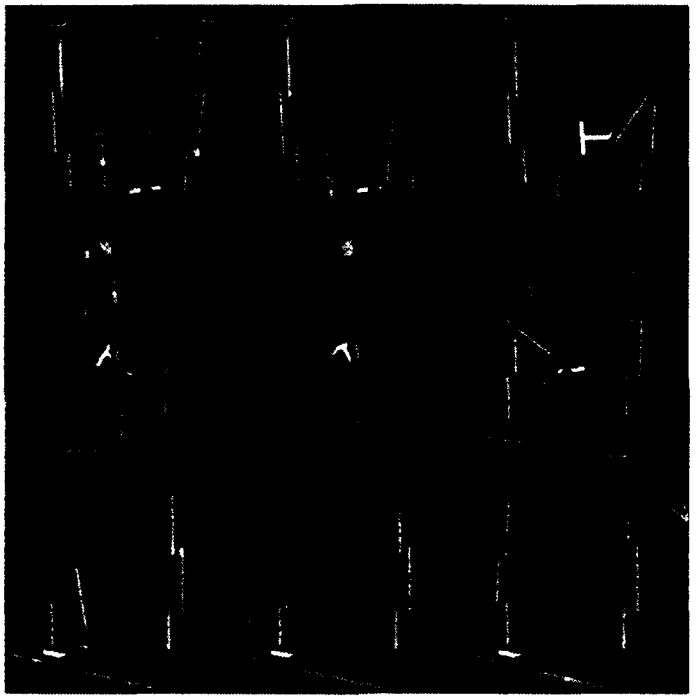


Figure 1. An example of a manipulation path. The object is T-shaped and requires only one arm to manipulate it.

An implementation of a new randomized manipulation planner was reported in the first and second quarter report of '93. This planner deals with a robot system consisting of three arms working in a three dimensional workspace. The object is manipulated with two arms (the object is assumed to be heavy, thus requiring two arms to manipulate it) and regrasping of the object is executed when necessary to ensure the completion of the task. In this quarter we have added some new features to the planner. The first improvement is the capability of planning manipulation paths for constrained object motions, for example the arm motions to turn a giant wheel. The other improvement is the possibility of using only one arm to manipulate an object but still allowing the arms to cooperate - in this case they pass the object from one arm to another thereby increasing the size of the workspace reachable by the object.

The strategy for finding the manipulation path is twofold. We first find an object motion that moves the object to the goal while satisfying constraints such as being a collision free path and that the arms are able to grasp the object. The second phase is to patch in the arm motions to actually grasp the object, to regrasp the object whenever a change of grasp occurs, and finally an arm motion to ungrasp the object once the goal is reached. To plan the manipulation paths for constrained object motions we then simply add these new conditions into the first phase of the planner. For example, in turning a giant wheel, the wheel must move such that it only rotates around its axis.

For finding the motions to manipulate an object using one arm, the change to the planner is simply in the description of the grasps for the object. We now specify that one arm is grasping the object rather than two. In our current version of the planner, this is achieved by editing an input file. An example path of one arm manipulation is shown in Fig. 1. The task is to manipulate the T-shaped object to the other side of the workspace. Notice that the first arm hands the object off to the second arm in order to complete the task. This particular path took approximately one and a half minutes to compute on a DEC alpha workstation.

4.5 Experiments in Manipulation Planning

In the last a few quarters, we have accomplished several preliminary experiments of our off-line manipulation planner with the dual-arm robot in the ARL under the distributed environment of Control Shell. The software modules in the system includes a hierarchical robot controller, a user interface, a simulator, a task planner and several path planners. The interface between these modules are also well defined and made easy through the NDDS (Network Data Delivery Service). The task planner in the first implementation only considers a single static object that requires two-arm cooperative manipulation. In this quarter, we started to extend our task planner to consider a more general case where multiple tasks can be specified by the user and multiple objects may appear in the work space at any time.

In the final demonstrative scenario, multiple objects that require either one arm or two arms can be mixed in the same work space. The task planning becomes more complicated since the motion of the two arms can be either synchronous when both arms are grasping the same object or asynchronous when they are manipulating different objects. We are extending the task planner to consider all these possible types of motion combinations in a more object-oriented fashion, i.e., different path planners are called depending on the type of the object and the status of the other arm.

We are also improving the failure handling function of the task planner. As mentioned in the previous reports, the motion plan generated by the path planner is a sequence of subpaths. They are converted into robot primitive commands (e.g. move arm A to configuration Q, close the gripper

of arm A, etc.) and sent to the robot controller one at a time. If the robot fails to executing a command (e.g. it fails to grasp an object), then the following commands would certainly fail. The task planner, however, should be able to detect the failure from the sensory data (e.g. the object doesn't move accordingly as the robot does) and update the commands according to the new status of the robot and the reason for the failure. This may requires partially or completely replanning depending on the reasons causing the failure. We are implementing this monitoring and failure handling function of the task planner in a state-machine-like scheme so that it can easily extended to the case where more objects and more arm motion types are involved.

More results on handling multiple objects and multiple tasks will be reported later.

4.6 Landmark-Based Mobile Robot Navigation

During the previous quarters we have developed and implemented several efficient algorithms for landmark-based mobile robot navigation. Mobile robot navigation is perhaps the most crucial problem in mobile robotics. Despite a lot of research effort over the past two decades, the problem still has no satisfactory solution. Prior theoretical studies and experiments with implemented systems tell us that:

- One cannot build a truly reliable system without both making clear assumptions bounding uncertainty and enforcing these assumptions by appropriately engineering the robot and/or its workspace.
- If assumptions are too mild, the planning subproblem is computationally intractable. If assumptions are too strong, engineering is too costly and/or navigation not flexible enough.

Our research investigates the tradeoff between "computational complexity" and "physical complexity" in reliable mobile robot navigation. Our approach consists of:

1. Defining a formal navigation problem with just enough assumptions to make it possible to construct a sound and complete planner that is also computationally efficient.
2. Designing and implementing such a planner, in order to verify that the planner is actually efficient.
3. Engineering a robot and its workspace to enforce the assumptions in the defined problem, in order to verify that the "cost" of such engineering is reasonable.
4. Implementing a navigation algorithm that makes a real robot execute plans generated by the planner, in order to verify that navigation is actually reliable.

This approach also induces a new role for experimentation in robotics: When robot algorithms are proven correct under formal assumptions, the purpose of experimentation shifts from demonstrating that they behave as intuitively expected on a sample of tasks, to verifying that the amount of engineering induced by the assumptions is acceptable.

Our previous quarterly reports describe work on steps 1 and 2 above. In the Spring'93 quarter we started dealing with steps 3 and 4. Our work centered on experimentations necessary to implement the landmark-based motion planner developed during the previous quarter. We decided to use visual landmarks that are to be located on the ceiling in an indoor environment to designate the landmark regions, where the mobile robot is assumed to have no uncertainty in sensing. A CCD camera module is installed on the top of the robot, pointing upward to detect and identify landmarks.

During this quarter we implemented a prototype of the landmark-based planning and navigation of a mobile robot in an indoor environment using visual landmarks located on the ceiling. We also improved the design of the landmark to allow larger number of landmarks (up to 512) and implemented a faster recognition algorithm (approximately 600 milliseconds on an 80386-based robot).

4.6.1 Implementation of Landmark-based Navigation

The robot, equipped with a CCD camera pointing vertically upward, executes the motion plan using the visual landmarks located on the ceiling. We used five landmarks in our Robotics Laboratory with four stationary, known obstacles such as a trash can and chairs.

Performance Evaluation and Limitations The robot successfully executed motion plans generated by the landmark-based planner. In order to ensure a reliable navigation, we needed to satisfy at least two conditions as follows.

1. A landmark cannot be located too close to illuminations on the ceiling. A direct illumination is likely to saturate the CCD camera and incapacitate the vision function. In our current experimental setting, we needed to locate landmark at least 10 inches away from an illumination.
2. The floor must be almost completely flat in the landmark region. A small hump could disorient the robot, hence tilting the camera with respect to the ground. As a result, the robot would not be able to localize with sufficient accuracy to navigate.

4.6.2 Improved Landmark Recognition

For the implementation discussed above, we used the landmark design developed in the previous quarter. The design presented two limitations.

1. Only six distinct landmarks can be used in an environment.
2. Detection alone takes more than 1 second and slows the travel speed of the robot.

We improved the landmark design and recognition algorithm to attenuate these limitations.

New Landmark Design A new landmark design is a black-on-white symbol. It consists of an outer envelope which resembles the letter C, 10-inches in diameter. The opening of C gives the symbol an orientation. Inside the letter C is a three-by-three square grid, aligned with the orientation of the landmark. Each grid cell is one inch by one inch and painted either white or black. Identification of the landmark is encoded in this binary grid, allowing up to $2^9 = 512$ landmarks.

New Recognition Algorithm We implemented a faster recognition algorithm specifically designed for the landmark design described above. The resulting algorithm runs in the order of $O(m+n)$, where m, n are the numbers of the pixels in the model and a given image, respectively. For comparison, the algorithm implemented in the previous quarter ran in $O(mn)$.

Recognition proceeds as follows. Using a statistical measure based on histogramming, the recognizer first finds the intensity threshold to binarize the gray-scale image into *white* and *black* pixels. The recognizer proceeds to find connected components (*blobs*) consisting of black pixels, discarding the ones whose size (measured in terms of the bounding rectangle) deviates too far from the expected size of a landmark. The surviving blobs are recorded as potential landmarks.

Observe that the pixel size of the landmark is known *a priori* since we are assuming the ceiling height to be constant at this point. Even if that assumption did not hold, we could change the physical size of the landmarks to accommodate lower or higher ceilings within a reasonable limit.

To detect a landmark from the blobs recorded as above, the recognizer overlays the black-and-white model of the outer envelope of the landmark on each blob region and counts the pixels whose binary values do not agree. A blob whose mismatch is below 40landmark. The 40account the grid code in the center which introduces extra black pixels. The geometric center of the bounding rectangle of the blob is recorded as the landmark location.

Once a landmark is found, the recognizer determines the (relative) orientation of the landmark by scanning the perimeter of the landmark in circular fashion to detect a break, or a concentration of white pixels.

The identity of the landmark is determined by decoding the binary grid. The recognizer examines two distinct pixel locations well inside each grid cell to decode the ID. If the two pixel values do not agree in any of the cells, the recognizer reports an error and dismisses the blob as a noise. We observed that any landmark that passes the overlay matching test is located very precisely (± 1 pixels). Thus, two distinct pixels in each cell must have the same values; otherwise, the blob is unlikely to be a landmark.

Performance The recognition process described above takes approximately 600 milliseconds on the robot, which runs on an Intel 80386 processor. Measurement revealed that the localization error was within ± 1 pixels, which translates to ± 0.22 inches in our experimental setup with an 8-foot ceiling. The orientation detection showed errors of ± 3 degrees. No landmarks have been misidentified.

4.7 Mobile Robot Navigation Toolkits

The main effort of this quarter has been focused on testing the toolkit modules that we have developed on a real robot, a Nomad 200 robot. In particular, we have developed and experimented a navigation system that consists of the following toolkit modules:

1. Approximate cell decomposition based motion planning module.
2. Artificial potential field based motion control module.
3. Sensor-based localization module.

We have performed extensive experimentation of this navigation system on a real robot. In addition, we have successfully integrated this navigation system with a cognitive level task planning system based on BB1 to produce a office surveillance robot system (cooperation with Dr. Barbara Hayes-Roth, Knowledge Systems Lab., Stanford). We have performed many experimentation of this office surveillance system, both in the robot simulator and with a real robot.

4.8 Simulator for Multiple Robots

In this quarter we have focused our effort in transferring the multiple robot simulation technologies to Nomadic Technologies. Nomadic Technologies is currently working on further developing these techniques and on integrating this multiple robot simulation capability into Nomadic Software Development Environment.

4.9 Summary of Main Results Obtained So Far

1. Identification of several axes for distributing path planning software in an on-line architecture.
2. A documented Randomized Path Planner package has been made available to other research institution on the computer network. Several organizations are using it.
3. Implementation of parallel versions of RPP on a Silicon Graphics 4D/240 multiprocessor machine and on a local-area network of UNIX-based workstations.
4. Definition of a new, FFT-based method to compute obstacles in configuration space.

5. Definition and implementation of a new path planning method (the vector-based planner) to generate paths for robots with many degrees of freedom.

6. Integration of several path planners (RPP, vector-based planner with/without potential fields) in a package distributed over a network of UNIX-based workstations.

7. Design and implementation of an optimal-time motion planner for closed-loop kinematic chains.

8. Design and implementation of a randomized three-arm manipulation planner for manipulating an elongated object in a 3D cluttered environment.

9. Design and implementation of a new landmark-based mobile robot planning method. Extension of this planner to deal with controllable uncertainty.

10. Definition of the layout of a software toolkit to efficiently develop new navigation systems. Implementation of several toolkits.

11. Partial development of a powerful multi-mobile-robot simulator to facilitate the development and debugging of programs for multiple interacting mobile robots.

Others:

- J.C. Latombe was elected AAAI Fellow for his contributions to the "Theory and Practice of Robot Motion Planning."

- C.Becker was a member (with 3 other students) of the Stanford team that won the first event at the AAAI-93 Mobile Robot competition, using our NOMAD 200 robot.

4.10 Status

Our research progresses according to schedule.

Chapter 5

Applications and Technology Transfer

It is not possible to develop generic technology without multiple, specific applications to test and refine the ideas and implementations. As such, we are actively seeking sites, both internally and externally to provide the compelling test beds that will make this project succeed. These driving applications span a variety of the most important target users: high-performance control, intelligent machine systems, underwater vehicle command and control, and remote teleoperation. Several of these projects will reach for new limits in advanced technology and system integration; others will address real-world problems in operational systems.

With the reduced funding levels, we will not have the resources to support all of the originally proposed technology evaluation sites. However, we believe these sites are crucial to the development of ControlShell into a viable technology for "real-world" use. Thus, we have actively pursued alternative means of supporting external sites. We have been successful in securing several new test applications. These sites will either function with minimal support, or fund their own support.

This chapter highlights some of the activities of these projects.

The currently-active ControlShell applications are:

- Precision Machining, by The Stanford Quiet Hydraulics Laboratory.
- Underwater Vehicle Control, a joint project between the ARL and the Monterey Bay Aquarium Research Institute.
- Intelligent Machine Architectures, by Lockheed Missiles and Space Corporation.
- Remote Teleoperation, by Space Systems Loral Corporation.
- Space-based Mobile Robot Systems, by several ARL students (NASA-sponsored).

- High-Performance Control of Flexible Structures, by several ARL students (AFOSR-sponsored).
- Space-structure assembly, by NASA Langley Research Center.
- Mobile-robot control by NASA Ames Research Center.

5.1 NASA sites

Partially as a result of the architectures seminar series, two NASA sites have committed to using ControlShell in their projects. The first site is a robotics laboratory at NASA's Langley Research Center. This laboratory will be studying robotic construction of space structures, using a single industrial robot. The work focuses on end-effector development, system integration, and tools to ease construction planning.

Another NASA lab, the intelligent mechanisms group at NASA Ames, is also using ControlShell for a mobile robotics project. This project's goals include studying intelligent exploration algorithms and robotic control architectures.

5.2 MBARI Underwater Vehicle

The Monterey Bay Aquarium Research Institute (MBARI), in a joint program with ARL, is using ControlShell to develop the control system of the OTTER (Ocean Technology Testbed for Engineering Research) vehicle. The vehicle is a unmanned submersible about 2 meters long and weighing 150 kilograms. It is powered by 8 thrusters and has an active vision sensing system. The vehicle will be used in research focusing on semi-autonomous control. On the vehicle, ControlShell is the the backbone of the control software that runs the thrusters and processes the sensor inputs. Control laws and data processing filters implemented in ControlShell allows us to feedback sensor input for automatic positioning and station-keeping.

The Finite-State Machine facility of ControlShell is used at the supervisory level to coordinate vehicle actions in a programmed response to stimuli. Stimuli can be produced by automatic sensing and error detection processes or through direct user interaction. FSM's are used to program complex vehicle actions to complete tasks autonomously.

The project uses NDDS for all interprocess/interprocessor communications. The two independent real-time computing systems on-board the vehicle are connected to a off-board real-time computer through a SLIP (Serial-Level Internet Protocol) line during operations. That provides network connectivity—albeit at a slower rate—even with the slow underwater acoustic modem communications. In addition, the graphical user interface running on an HP workstation is connected via Ethernet to the off-board real-time system. NDDS provides the basic communications and message passing between all four independent systems. Since NDDS is network transparent, any of the

computing systems can be replaced with a simulation during development to test software and other components of the entire OTTER system.

5.3 Transfer of Planning Technology

- Part of our mobile robot software (simple planner, simulation) has been ported by Nomadic Technologies, and is part of the software distributed by this company with their mobile robot NOMAD 200.
- J.C. Latombe and L. Kavraki assisted Nova Management, Inc., in building an automated route planner for tanks in support of US Government Contract No. DAAE07-C-93-0026. A prototype version of this planner was successfully demonstrated to Army representatives.
- B.Romney (a PhD student) spent the summer at GM Research Labs in Warren, MI, and implemented a version of assembly planner there. He connected this planner to the UNIGRAPHICS CAD system.
- R.H. Wilson (a Ph.D. student, then a Research Associate) was hired as a Research Scientist by SANDIA Labs, Albuquerque, NM.

Publications So Far:

R.I. Brafman, J.C. Latombe, and Y. Shoham, "Towards Knowledge-Level Analysis of Motion Planning," *Proc. of the 11th Nat. Conf. on Artificial Intelligence*, AAAI-93, Washington D.C., July 1993, pp. 670-675.

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform*, Technical Report, STAN-CS-92-1425, 1992.

L. Kavraki, *Computation of Configuration-Space Obstacles Using the Fast Fourier Transform*, *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Atlanta, GA, 1993.

L. Kavraki, J.C. Latombe, and R.H. Wilson, "On the Complexity of Assembly Partitioning," accepted for publication in *Information Processing Letters.*

L. Kavraki and J.C. Latombe, *Randomized Preprocessing of Configuration Space for Fast Path Planning*, Rep. No. STAN-CS-93-1490, Dept. of Computer Science, Stanford University, September 1993.

Y. Koga and J.C. Latombe, "Experiments in Dual-Arm Manipulation Planning," *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Nice, May 1992, pp. 2238–2245.

Y. Koga, T. Lastennet, J.C. Latombe, and T.Y. Li, "Multi-Arm Manipulation Planning," *Proc. of the 9th Int. Symp. on Automation and Robotics in Construction*, Tokyo, June 1992.

J.C. Latombe, "Geometry and Search in Motion Planning," *Annals of Mathematics and Artificial Intelligence*, 8(2-4), 1993.

J.C. Latombe, "Robot Algorithms," *Proc. of the LAAS/CNRS 25th Anniversary Conf.*, Cepadues, Toulouse, France, May 1993, pp. 81–94 (invited conference).

A. Lazanas and J.C. Latombe, *Landmark-Based Robot Navigation*, Rep. No. STAN-CS-92-1428, Dept. of Computer Science, Stanford U., May 1992. Accepted for publication in *Algorithmica.*

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Navigation," *Proc. of the 10th Nat. Conf. on Artificial Intelligence, AAAI-92*, San Jose, July 1992, pp. 816–822.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Proc. of the AAAI Fall Symp.*, Boston, MA. October 1992, pp. 98-103.

A. Lazanas and J.C. Latombe, "Landmark-Based Robot Motion Planning," *Geometric Reasoning for Perception and Action*, C. Laugier (Ed.), Lecture Notes in Computer Science, 708, Springer-Verlag, 1993.

Gerardo Pardo-Castellote and Robert H. Cannon Jr. "Proximate time-optimal parameterization of robot paths," STAN-ARL-92- 88, Stanford University Aerospace Robotics Laboratory, April 1993.

Gerardo Pardo-Castellote, Tsai-Yen Li, Yoshihito Koga, Robert H. Cannon Jr., Jean-Claude Latombe, and Stan Schneider," "Experimental integration of planning in a distributed control system. In *Preprints of the Third International Symposium on Experimantal Robotics*, Kyoto Japan, October 1993.

Gerardo Pardo-Castellote and Stanley A. Schneider. "The network data delivery service: real-time data connectivity for distributed control applications," (to appear in) In *Proceedings of the International Conference on Robotics and Automation*, San Diego, CA, May 1994. IEEE, IEEE Computer Society.

Gerardo Pardo-Castellote and Stanley A. Schneider. "The network data delivery service: A real-time data connectivity system," (to appear) In *Proceedings of the AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service and Space*, Houston, TX, March 1994. AIAA, AIAA.

S. Schneider and R. H. Cannon. "Object impedance control for cooperative manipulation: Theory and experimental results," *IEEE Journal of Robotics and Automation*, 8(3), June 1992. Paper number B90145.

S. A. Schneider and R. H. Cannon. "Experimental object-level strategic control with cooperating manipulators," *The International Journal of Robotics Research*, 12(4):338–350, August 1993.

Howard H. Wang, Richard L. Marks, Stephen M. Rock, and Michael J. Lee. "Task-based control architecture for an untethered, unmanned submersible," In *Proceedings of the 8th Annual Symposium of Unmanned Untethered Submersible Technology*, pages 137–147. Marine Systems Engineering Laboratory, Northeastern University, September 1993.

DFE Editor

Revision 0.1
The ControlShell Data-Flow Graphical Editor

1. Introduction

The **DFE Editor** is a graphical tool for creating and viewing block diagrams of your real-time *ControlShell* system. You specify the *ControlShell* components (execution modules) and the signals that flow between the components. The **DFE Editor** generates data files that can be directly used in *ControlShell* applications[1]; *ControlShell* utilizes the data-flow input-output relationships to sort and arrange the execution order for each component. You can also use the simple point-and-click interface of this graphical editor to define named *Categories* and *Module Groups* to which a component belongs.

The basic graphical objects in a **DFE** diagram are components, signals, and labels. The components are represented by rectangles with input, output, and reference "pins"; the signals are represented by lines; and the labels are represented by text. An example of a data-flow diagram is show in Figure 1.

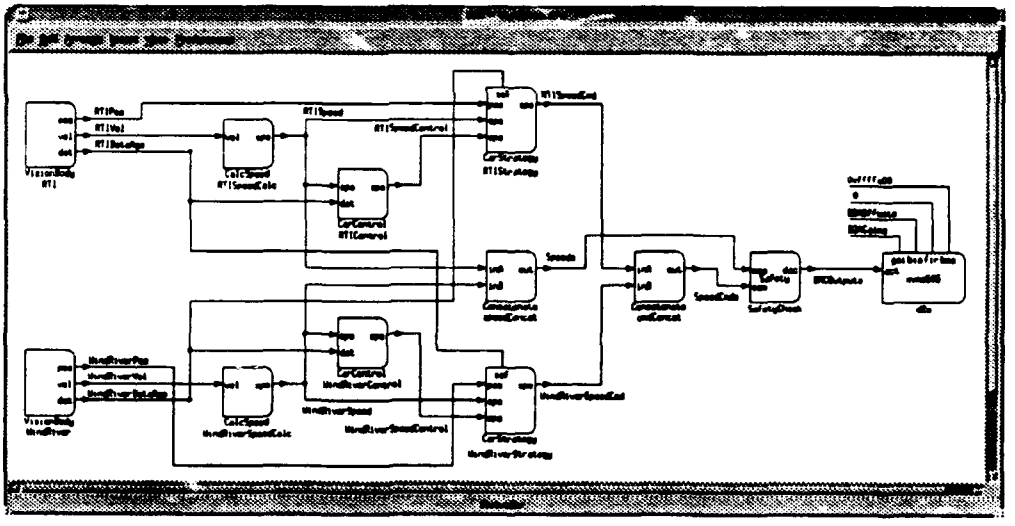


Figure 1 Data-Flow Diagram Example

Labels can be used to add notes, but are generally bound to components to provide an instance name for each component, and bound to signals to provide signal names.

This document assumes a working knowledge of *ControlShell* components and just describes the steps involved in creating and maintaining data-flow diagrams.

[1] Use the `CSDfeParseFile()` C-function to have *ControlShell* read and process **DFE** files.

1.1. Before You Begin

DFE Editor requires a resource file called **Dfe** in the **app-defaults** directory. This file specifies all the menu text, quick-keys and the translations from mouse events to drawing actions. It also specifies resources such as colors and grid size. The default **Dfe** file may be found in the **/local/applications/rti/app-defaults** directory. You should copy it to your personal **~/app-defaults** directory or a global app-defaults directory set up by your system administrator.

Set the **XUSERFILESEARCHPATH** environment variable to make sure the **DFE Editor** can find the resource file. For example, type at the UNIX prompt:

```
setenv XUSERFILESEARCHPATH ~/app-defaults/%N%S
```

Also make sure the **LD_LIBRARY_PATH** includes the **X11R5** directory. Run:

```
setenv | grep LD_LIBRARY_PATH
```

If the result does not include the **X11R5** directory, and your **X11R5** libraries are located in **/local/X11R5/lib/sun4**, type:

```
setenv LD_LIBRARY_PATH /local/X11R5/lib/sun4:$LD_LIBRARY_PATH
```

You also need to define an environment variable to indicate the directories where the **DFE Editor** can find the component definition files (**.ce** files). Separate each directory with a colon:

```
setenv CS_CELOADPATH /local/applications/rti/cs/lib/celib:
${HOME}/lib/components
```

If you have many component directories, it might be better to set up a single directory and use symbolic links to point to the actual component definition files. There is a limit to length of an environment variable[2]. The **DFE Editor** will always search the current directory before the **CS_CELOADPATH**.

1.2. Starting the DFE Editor

To start the **DFE Editor**, create and change to a directory that is to hold **DFE** data files. Type **dfe&** at the UNIX prompt. You will be presented with the initial drawing screen.

2. Creating and Editing a Data-Flow Diagram

The **DFE Editor** allows a Data-Flow block diagram to be created with simple mouse actions, and relatively familiar drawing motions. Only the left and middle mouse buttons are used at this time, in conjunction with the **<Shift>** and **<Ctrl>** keys.

The menu bar contains several main menus: **File**, **Edit**, **Arrange**, **Insert**, **View**, and **Preferences**. The **File** menu provides access to files. The **Edit** menu provides additional editing commands, such as **Undo**, **Delete**, and **Duplicate**. The **Arrange** menu contains commands to rotate components in 90° increments. The **Insert** menu provides a way of inserting components and labels. The **View** menu provides zoom capability. The **Preference**

[2] In a future release, the paths can be defined in a .dfe-init file.

menu provides commands for toggling snap-to-grid mode and auto-save mode.

The tables in the following subsections further describe mouse actions and the menu options.

2.1. Using the Mouse

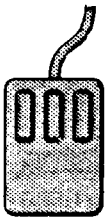
This section describes how the mouse is used.

2.1.1. Drawing using the Mouse—Middle Button

The drawing actions associated with the middle mouse button are defined in Table 1.

Table 1 Using the Middle Mouse Button

Middle Mouse Button	Action
Click-Drag	Create a signal. The "root" of a signal carries a label defining the signal name. If the starting point is near an output end of an existing signal, a new segment is added to the signal. If the starting point is near an existing signal line segment, a new branch is added to the signal.
<Ctrl>-Click-Drag	Create a label.



When a signal is created, a default signal name is added of the form:

Signal*n*

where *n* is an integer that increments as more signals are created.

You must double-click the left mouse button on a signal name to change it.

When a component is created, it is assigned a name of **Component***n* where *n* is an integer that increments as more components are created.

Similarly, when a label is created, it is assigned the text, **Label***n*.

2.1.1.1. Signal-Name Label

For each signal there is a blue label representing the signal name. It is located at the "root", or start, of each signal tree. It must be a single word.

2.1.1.2. Component-Name Label

The name of a component can contain any alphanumeric character and underscore (_), but it must be a single word. No white space is allowed.

2.1.1.3. Label Text

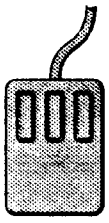
A label text may contain any text and may be any number lines. It is always center aligned.

2.1.2. Editing using the Mouse—Left Button

The actions associated with the left mouse button are defined in Table 2.

Table 2 Using the Left Mouse Button

Left Mouse Button	Action
Click	Deselect all objects, then select the object within range (pick radius) of the mouse pointer.
<shift>-Click	Toggle the selection of the object within range of the mouse pointer.
Click-Drag	If the mouse pointer is within range of a selected object, all selected objects are moved. Otherwise, start a selection rectangle. When the button is released, all objects completely within the selection rectangle will be selected.
Double-Click	If the mouse pointer is within range of a Label object, an edit dialog box is popped up for changing the label text. Otherwise, no action.



A signal is composed of "nodes". Generally, each node is where a signal bends or branches. The user can move individual nodes of a signal or line segments. To select only a node, first deselect the signal, then click near one of the nodes; active nodes are represented by small black squares. To select a line segment, click on the segment or **<shift>**-Click on the two nodes that make up the segment.

If a component is selected, all signal segments attached to it will also be selected. *Be careful when deleting after selecting a Component!* A signal is "attached" to a component if one of the ends coincides with the end of a component pin. Signal nodes and component pins snap to a grid (default 20 pixels), easing the task of attaching signals to pins. Label objects do not snap; it makes them easier to place in the drawing window.

When editing a signal or component name, the dialog box will prevent any white space to be inserted, forcing a single word.

2.2.Using the Menus

Using the menus, the user can open and save DFE files, duplicate and delete objects, and insert DFE objects. The menus are separated into six (6) categories, **File**, **Edit**, **Arrange**, **Insert**, **View** and **Preference**. The following sections describe the functions of each menu command.

If the menu command has a QuickKey combination, it is noted in parentheses after the command name.

2.2.1. File Menu (Meta+F)

The **File** menu performs file operations to save and retrieve DFE files.

2.2.1.1. New (Ctrl+N)

Choosing **File**, **New** will clear the DFE Window, prepared to start a new data-flow diagram. If the current diagram has been edited, the user will be prompted to save it first.

2.2.1.2. Open (Ctrl+O)

Choosing **File**, **Open** will retrieve an existing DFE definition from a file. If the current diagram has been edited, the user will be prompted to save it first. The editor will then pop up the **File**, **Open** selection box, as show in Figure 2.

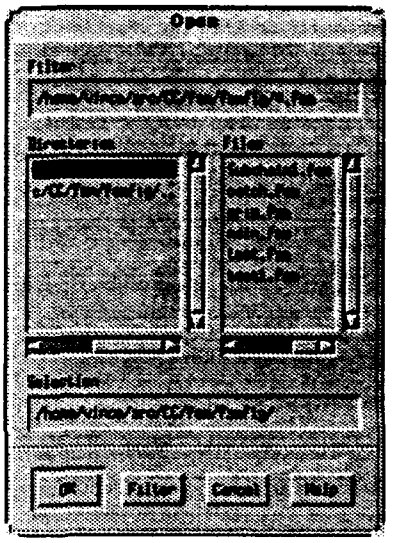


Figure 2 File-Open Selection Box

Use the **Directories** list box to move between directories by double-clicking on the desired directories. To quickly jump to another directory, you can replace the filter string. For example, to jump to the

/local/applications/rti/dfe

directory to look for DFE files, replace the string in the **Filter** text box with:

/local/applications/rti/dfe/*

and hit **<Enter>** or click the **Filter** button. Do not forget the "*" wildcard.

To choose a file, double-click on the file name in the **Files** list box, or highlight the filename and click **OK**.

Parsing Errors

The parser checks for errors in the DFE file when it is opened. If there is an error, a dialog pops up to ask if the file should be re-read. This gives the user a chance to edit the file before continuing. The UNIX command window from which **dfe** was invoked will contain the line number at which the error occurred. Generally, you will want to check to make sure your **CS_CELOADPATH** environment variable is set correctly so that the **DFE Editor**

can find the components referenced in the DFE file. If not, you will need to exit, set **CS_CELOADPATH**, and restart **dfe**.

Although re-reading will confirm that errors have been removed, there may be extra objects left in the DFE window. It is best to call **File**, **Open** again.

2.2.1.3. Save (Ctrl+S)

Choosing **File**, **Save** saves the current drawing window.

If the current window has not been saved before (or was not read from a file), the **File**, **Save As** selection box pops up to ask for a new file name. If the selected file already exists, the user will be asked to confirm overwriting the old file.

Backup File

File, **Save** saves a backup file with a file name that contains an additional **.bak** extension.

2.2.1.4. Save As (Ctrl+A)

Choosing **File**, **Save As** saves the current DFE under a new name. It will prompt for the name of the *sample habitat* to which the components belong.

2.2.1.5. Generate EPS... (Ctrl+E)

File, **Generate EPS** creates an Encasulated PostScript (EPS) file of your DFE diagram. You can easily import this file into your documentation files.

You will be prompted for a file name, and if you leave out the file extension, **.eps** will be appended.

Most document preparation applications will adjust the size of the EPS figure to fit inside the document. If your DFE drawing is large, the figures may be reduced too much to be legible. In these cases, you may want to invoke special scaling and cropping features of your document-preparation application to display portions of the EPS file.

To change the properties of the generated EPS file, such as printer resolution and fonts, make the changes in the **Dfe** app-defaults file.

2.2.1.6. Quit (Meta+Q)

File, **Quit** quits the **DFE Editor** application. If the drawing window has not been saved, the user will be prompted to confirm saving. If, in addition, the drawing window has never been saved, the **File**, **Save As** selection box is popped up to ask for a new file name.

2.2.2. Edit Menu (Meta+E)

The edit menu provides some rudimentary editing commands.

2.2.2.1. Undo (Meta+Bksp)

Choosing **Edit**, **Undo** can undo some commands. If there is an action that can be undone, the **Undo** button will be enabled. Otherwise, the button is "greyed out". Some of the commands that may be undone are:

- Creation of objects (Component, Signal, Label)
- Deletion of objects (via **Edit**, **Delete**)

You cannot undo moves or file operations.

2.2.2.2. Duplicate (Ctrl+D)

Duplicate is used to duplicate a component or label. Signals cannot be duplicated in this release.

2.2.2.3. Delete Node

Delete the active line nodes. If the node at the root of a signal is selected, the entire signal tree (all branches) are delete. Use this command to clean up unnecessary nodes in signal lines.

2.2.2.4. Delete (Bksp)

Choosing **Edit**, **Delete** will delete all selected objects.

Be careful when deleting components, since selecting a component also selects signals attached to it. To delete a component without deleting the attached signals, use **<Shift><LeftClick>** to toggle OFF the selection of the signal nodes.

Additionally, if the signal-name label is deleted, the entire signal (all branches) is deleted.

Similarly, if the component-name label is deleted, the component is also deleted.

2.2.2.5. Category (Ctrl+C)

Choosing **Edit**, **Category** calls up the Category Dialog Box, as shown in Figure 3. From here, you can create and edit Categories and Module Group names. To place components into a Module Group, simply highlight the Module Group name and click on the components. Each click on a component toggles its inclusion in the Module Group.

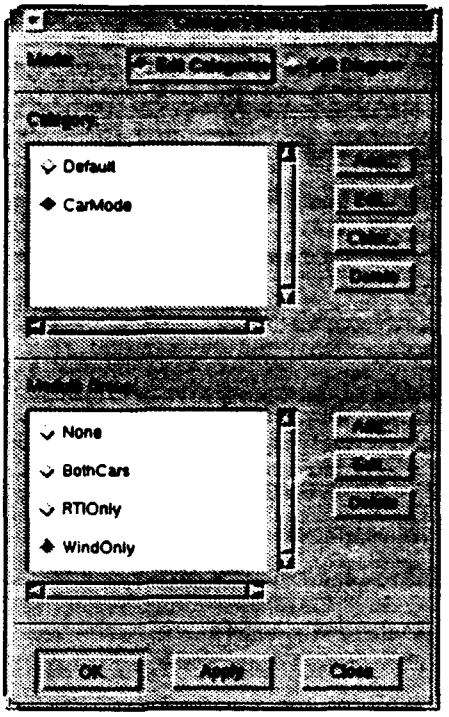


Figure 3 Edit-Category Dialog Box

You can also define the color associated with each Category. A component of that Category will be drawn in the chosen color, letting you tell at a glance how the components are related in your diagram.

2.2.3. Arrange Menu (Meta+A)

The **Arrange** menu contains commands to rotate selected components in 90-degree increments. Unfortunately, connected signals do not follow rotated components.

2.2.4. Insert Menu (Meta+I)

The **Insert** menu provides a way of creating various DFE objects. It always places the new object in the upper-left corner of the drawing window.

2.2.4.1. Component (Meta+C)

Choosing **Insert**, **Component** creates a new Component. You are presented with a file-selection box from which you can select the appropriate (.ce) component-description file. Along with the list of current subdirectories, the **Directories** list box lists the directories in your **CS_CELOADPATH** .

2.2.4.2. Label (Meta+L)

Choosing **Insert**, **Label** creates a new label object. Using this may be quicker than using the mouse, and is easier to remember than **<Ctrl><MiddleButtonDrag>**.

2.2.5. View Menu (Meta+V)

The **View** menu contains zoom factors for displaying the drawing: 25%, 50%, 75%, 100%, 125%, 150%, 200%. In the current version, the text in the DFE Editor is not scaled, so positioning will appear correctly only at 100% zoom. Below 75% zoom, all text will be represented by a single period (.).

2.2.6. Preferences Menu (Meta+P)

The **Preferences** menu allows the setting of user preferences for the DFE Editor.

2.2.6.1. Snap-to-Grid Mode (Ctrl+G)

Toggles the snap-to-grid behavior when drawing and placing components and signals. The grid size cannot be set from within the editor. You must set the associated resource in the **app-defaults** file or in **.Xdefaults**:

```
Dfe*Stage.grid:   20
```

2.2.6.2. Auto-Save Mode

Toggles auto-saving of the current DFE drawing. The auto-save interval is specified in the **app-defaults** file in units of minutes:

```
Dfe*autosaveInterval:   2
```

The default interval is 5 minutes.

The autosave file has the same name as the current file, with an appended '#' character. Thus, the auto-save file for **main.dfe** is **main.dfe#**. After the file is explicitly saved the auto-save file is deleted.

If the current file has never been saved, Auto-Save will pop up the **File, Save As** file-selection box to prompt for a file name.

3. Data-Flow Diagram Objects

The Data-Flow diagram objects consist of components and signals. The sub-sections that follow describe the objects in more detail.

The **DFE Editor** will allow you to save and retrieve partially completed DFE files, letting you work at your own pace, but *ControlShell* will not be able to successfully parse these incomplete DFE files at run-time. When you save an DFE, the **DFE Editor** will warn you of the inconsistencies.

3.1. Component

A *ControlShell* component may have any number of input, output and reference signal "pins". You write the actual execution code for each component, but you define the relationship between components using the **DFE Editor**.

A component is represented by a rectangle on the drawing screen, with a string giving the name of the component. The input pins are represented by short line segments with arrows pointing into the rectangle; the output pins are represented by line segments with arrows pointing out of the rectangle; and reference pins do not have arrow heads. Each pin is also labeled with

the first three (3) letters of its pin name. Finally, an editable label represents the instance name of the component.

Occasionally, the three letters are not enough to distinguish the pin names. You'll need to consult the corresponding `.ce` file to resolve the differences. The pins are drawn in the order they are listed in the `.ce` file, so it's not too difficult to figure out.

When a component is selected by the left mouse button, all attached signals are also selected. This allows you to move a component and still retain its connections. If you do not want to move the nodes, **<Shift><LeftClick>** each node to toggle OFF its selection.

3.2. Signal

A signal has a "root" to which the signal-name label is attached. This root is an artificial definition and does not indicate anything about signal flow. The component pins define flow direction, and the signal lines just serve to indicate which pins are connected to each other.

In fact, unconnected signals of the same name will be treated as the *same* signal.

The current implementation does not allow you to physically connect two existing signals. To create branches in a signal to connect to multiple places, you must draw a signal line (click and drag the middle mouse button) starting near an existing line segment or node. You add segments to any signal by drawing a line starting from any endpoint *except* the root node (the one with the signal-name label attached.)

3.3. Label

A label object is just a place where you can add text or comments. It can be multi-lined, but is always center aligned.

4. Data-Flow Editor Data Files

The data file used by the **DFE Editor** is the same as that used by the *ControlShell* run-time parser. The only additions are coordinates and color specifications for the drawing objects.